# A Robust Parallel Adaptive Mesh Refinement Software Library for Unstructured Meshes

John Z. Lou, Charles D. Norton, and Tom Cwik
Jet Propulsion Laboratory
California Institute of Technology, Pasadena, CA 91009-8099

**Abstract**

The design and development of a software tool for performing parallel adaptive mesh refinement in unstructured computations on multiprocessor systems are described. This software tool can be used in parallel finite element or parallel finite volume applications on triangular and tetrahedral meshes. It contains a suite of well-designed and efficiently implemented modules that perform operations in a typical P-AMR process. This includes mesh quality control during successive parallel adaptive mesh refinement, typically guided by a local-error estimator, and parallel load-balancing. Our P-AMR tool is implemented in Fortran 90 with a Message-Passing Interface (MPI) library, supporting code efficiency, modularity and portability. The AMR schemes, Fortran 90 data structures, and our parallel implementation strategies are discussed in the paper. Test results of our software, as applied to selected engineering finite element applications, will be demonstrated. Performance results of our code on Cray T3E, HP/Convex Exemplar parallel systems, and on a PC cluster (a Beowulf-class system) will also be reported.

**Keywords: parallel adaptive mesh refinement, unstructured mesh**

## 1. Introduction

Adaptive mesh refinement (AMR) represents a class of numerical techniques that has demonstrated great effectiveness for a variety of computational applications including fluid dynamics, structural mechanics, electromagnetics, and semiconductor device modeling. However, the development of an efficient and robust adaptive mesh refinement component for an application, particularly on unstructured meshes on multiprocessor systems, involves a level of complexity that is beyond the technical domain of most computational scientists and engineers. The motivation for our work is to provide an efficient and robust parallel AMR library that can be easily integrated into unstructured parallel applications.

Research on parallel AMR for unstructured meshes has been previously reported [2,10]. Most efforts are C++-based, and many realize that mesh quality control issues during successive adaptive refinement is an active research topic. The features of our work include the design of a complete Fortran 90 data structure set designed for parallel AMR on unstructured triangular and tetrahedral meshes, and a robust scheme for mesh quality control during the parallel AMR process. We selected Fortran 90 for our implementation mainly because it provides adequate facilities for parallel unstructured AMR development while simplifying interface concerns with scientific application codes, many of which were developed in Fortran 77.

Mesh quality control is an important issue that should be addressed in any AMR algorithm. If adaptive refinement on a mesh is guided by a local-error estimator [1] and AMR is performed repeatedly, a straightforward adaptive refinement scheme would usually result in mesh elements with poor aspect-ratios. Different strategies have been proposed to improve the quality of meshes generated from AMR. These include mesh smoothing after each adaptive refinement, selec-

tion of element refinement patterns based on the element shape, and other approaches [4]. We will present a robust approach to addressing the issue of mesh quality control during successive mesh refinement, discuss our implementation scheme for this technique and show some test results.

Our adaptive mesh refinement algorithm consists of two steps: a logical/conceptual step in which the information needed to refine each element in the coarse mesh is constructed and stored, and a physical refinement step in which the coarse mesh is actually refinement to produce a new mesh. The separation of an adaptive refinement process into a logical refinement phase and a physical refinement phase offers several advantages in a parallel AMR implementation. It makes the AMR code highly modular, and makes the actual mesh refinement local to each element. It also makes it possible to perform parallel load-balancing by migrating only the coarse mesh instead of the refined mesh, thus with a much reduced communication cost. Such a refinement strategy also makes it possible to confine the interprocessor communication to the logical refinement phase. The code for this phase is small compared to the actual refinement phase which is basically an operation local to each processor in the parallel AMR process.

## 2. Parallel mesh refinement

Our parallel AMR framework is composed of the following components: a parallel adaptive mesh refinement module for triangular meshes and tetrahedral meshes, a parallel graph partitioner for partitioning (weighted) graphs, and a mesh migration module that moves portions of a partitioned mesh to their assigned processors. In a typical parallel AMR application initially the input mesh is randomly distributed among the processors after loading from a disk file. This mesh is then partitioned by the graph partitioner and redistributed among the processors, by the mesh migration module, according to the new partitioning. After an application computation and a local-error estimate step, the mesh refinement module performs a logical AMR on the distributed mesh based on the local-error estimate in each mesh element. Since the outcome from the logical AMR indicates the distribution of computational load among processors for the refined mesh, a load balancing decision can be made. The physical AMR is performed after the load-balancing step, where a new mesh is created by refining a subset of elements in the coarse mesh. As the last step in the parallel AMR loop, the refined mesh may be checked for element quality, and an optional mesh smoothing procedure can be performed. A control diagram of our parallel AMR algorithm is shown is Figure 1.

The logical refinement step uses an iterative procedure that traverses through elements of the coarse mesh repeatedly to "define" a consistent mesh refinement strategy on the coarse mesh. The result of the logical refinement is stored in the coarse mesh element data structure which completely specifies whether and how each element in the coarse mesh should be refined. Our adaptive refinement scheme is based on "edge-marking" for both triangular and tetrahedral meshes. It proceeds by marking (or logically refining) element edges wherever necessary, and the refinement pattern for each element is determined by the number of marked edges in that element.

To perform the logical refinement in parallel, we extend the serial iterative procedure so that after traversing the element set for edge-marking, each processor updates the status of edges on mesh partition boundaries by exchanging boundary edge information with its neighboring processors. The iteration stops when no processor can find any more local edges to mark in a particular sweep through its local elements. Using this strategy, the serial mesh refinement module can be parallelized by only inserting a couple of message-passing calls, with no change to the rest of the serial code.

One problem associated with repeated AMR is the deterioration of mesh element quality. Since

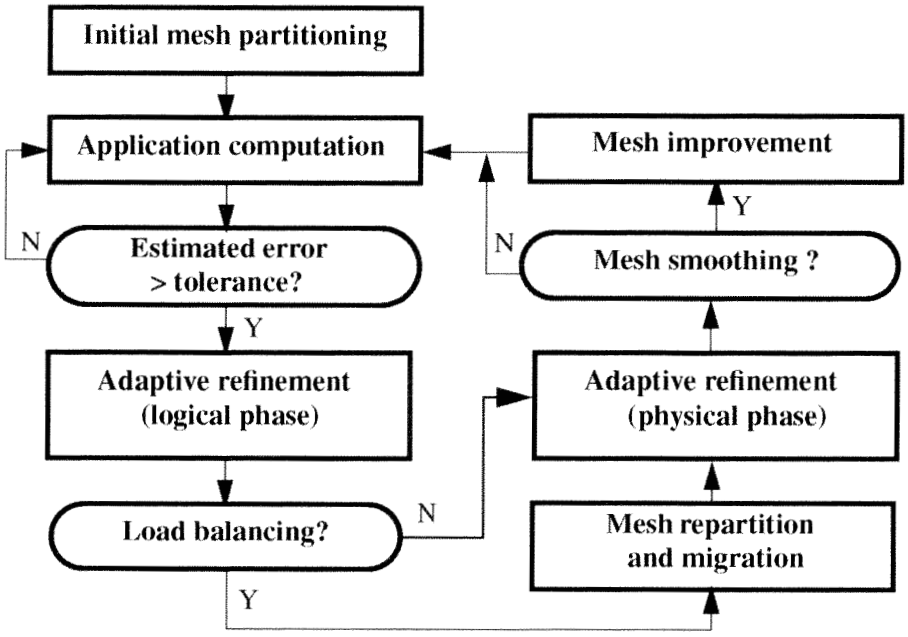


Figure 1. Parallel AMR process for unstructured meshes


Figure 2. An example of mesh quality control. The original refinement (left) on the coarse element 2-3-4 is modified (right) if any of the two elements in 2-3-4 need to be further refined either due to local errors or because their neighboring elements in 1-2-3 are to be further refined.

elements in the mesh are not uniformly refined (in order to preserve the consistency of the global mesh) the aspect-ratio of partially refined elements could degrade rapidly as adaptive refinement proceeds, especially for the three-dimensional tetrahedral meshes. Mesh smoothing algorithms have been proposed [3,7] to improve elements shape either locally or globally. Most mesh smoothing schemes tend to change the structure of the input mesh to achieve the "smoothing effect" by rearranging nodes in the mesh. The changes made by a smoothing scheme, however, could modify the desired distribution of element density produced by the AMR procedure. With a maximally refined mesh, applying a smoothing operation over the entire mesh is probably the only choice to improve the mesh quality. On the other hand, it is possible to prevent the mesh quality from further degradation during repeated adaptive refinement. The idea is to change the original refinement on a partially refined element if any of the children of that element need to be further refined in the next refinement. Figure 2 illustrates an example of the situation.

To simplify the implementation of such a feature in a parallel adaptive refinement procedure, we require that the mesh partitioner does not separate the twin elements (2-3-4) onto two processors, allowing the subsequent refinement operation to remain local in each processor. By incorporating this quality control

capability into the AMR procedure, the final mesh successive AMR stages will have an acceptable quality level if the initial input mesh does.

## 3. Fortran 90 implementation

Fortran 90 modernizes traditional Fortran 77 scientific programming by adding many new features. These features allow programs to be designed and written at a higher level of abstraction, while increasing software clarity and safety without sacrificing performance. Fortran 90's capabilities encourage scientists to design new kinds of advanced data structures supporting complex applications. (Developing such applications, like P-AMR, might not have been considered by Fortran 77 programmers since development would have been too complex.) These capabilities extend beyond the well-known array syntax and dynamic memory management operations.

Derived-Types, for instance, support user-defined types created from intrinsic types and previously defined derived-types. Dynamic memory management in the form of pointers, and a variety of dynamic arrays, are also useful in data structure design. Many of the new Fortran 90 data types know information about themselves, such as their size, if they have been allocated, and if they refer to valid data. One of the most significant advances is the module, that supports abstraction modeling and modular code development. Modules allow routines to be associated with derived types defined within the module. Module components can be public and/or private leading to the design of clean interfaces throughout the program. This is very useful when multiple collaborators are contributing to the design and development of large projects. Other useful capabilities include function and operator overloading, generic procedures, optional arguments, strict type checking, and the ability to suppress the creation of implicit variables. Fortran 90 is also a subset of HPF and, while message passing programming using MPI is possible with Fortran 90, this link to HPF simplifies extensions to data parallel programming. Both Fortran 90 and HPF are standards supported throughout industry which helps promote portability among compilers and machines.

One of the major benefits of Fortran 90 is that codes can be structured using the principles of object-oriented programming [6,9]. While Fortran 90 is not an object-oriented language, this methodology can be applied with its new features. This allows the development of a PAMR code where interfaces can be defined in terms of mesh components, yet the internal implementation details are hidden. These principles also simplify handling interlanguage communication, sometimes necessary when additional packages are interfaced to new codes. Using Fortran 90's abstraction techniques, for example, a mesh can be loaded into the system, distributed across the processors, the PAMR internal data structure can be created, and the mesh can be repartitioned and migrated to new processors (all in parallel) with a few simple statements as shown in Figure 3.

A user could link in the routines that support parallel adaptive mesh refinement then, as long as the data format from the mesh generation package conforms to one of the prespecified formats, the capabilities required for PAMR are available. We now describe the Fortran 90 implementation details that make this possible.

### 3.1 Fundamental Data Structures

Automated mesh generation systems typically describe a mesh by node coordinates and connectivity. This is insufficient for adaptive mesh refinement. Hierarchical information, such as the faces forming an element, the edges bounding each face, or elements incident on a common node is also useful. Additionally, large problems require the data to be organized and accessible across a distributed memory parallel computing system. These issues can be addressed by the creation of appropriate PAMR data structures.

```
PROGRAM pamr
use mpi_module ; use mesh_module ; use misc_module
implicit none
integer :: ierror
character(len=8) :: input_mesh_file
type (mesh) :: in_mesh
call MPI_INIT( ierror )
   input_mesh_file = mesh_name( iam )
   call mesh_create_incore( in_mesh, input_mesh_file )
   call mesh_repartition( in_mesh )
   call mesh_visualize( in_mesh, "visfile.plt" )
   call MPI_FINALIZE( ierror )
END PROGRAM pamr
```

Figure 3. A main program with selected PAMR library calls.

The major data structure is the description of the mesh. While a variety of organizations are possible, where trade-offs between storage and efficiency of component access must be decided, most descriptions include elements, faces, edges, and nodes. These are related hierarchically where components generally contain references to other components that comprise its description. These references can be bi-directional, an edge may have references to its two node end points and references to the faces it helps form. However, some of these details can be omitted from the structure by using a combination of good data structure designs and efficient recomputation of the required information.

**Basic mesh definition**

Fortran 90 modules allow data types to be defined in combination with related routines. In our system the mesh is described, in part, as shown in Figure 4. In two-dimensions, the mesh is a Fortran 90 object containing nodes, edges, elements, and reference information about non-local boundary elements (r_indx). These components are dynamic, their size can be determined using Fortran 90 intrinsic operations. They are also private, meaning that the only way to manipulate the components of the mesh are by routines defined within the module. Incidentally, the remote index type r_indx (not shown) is another example of encapsulation. Objects of this type are defined so that they cannot be created outside of the module at all. A module can contain any number of derived types with various levels of protection, useful in our mesh data structure implementation strategy.

All module components are declared private, meaning that none of its components can be referenced or used outside the scope of the module. This encapsulation adds safety to the design since the internal implementation details are protected, but it is also very restrictive. Therefore, routines that must be available to module users are explicitly listed as public. This provides an interface to the module features available as the module is used in program units. Thus, the statement in the main program from Figure 3:

```
call mesh_create_incore( in_mesh, input_mesh_file )
```

is a legal statement since this routine is public. However the statement:

```
element_id = in_mesh%elements(10)%id
```

is illegal since the "elements" component of in_mesh is private to the derived type in the module.

The mesh_module uses other modules in its definition, such as the mpi_module and the heapsort_module. The mpi_module provides a Fortran 90 interface to MPI while the heapsort_module is used for efficient construction of the distributed mesh data structure. The routines defined within the con-

```
module mesh_module
use mpi_module ; use heapsort_module
implicit none
private
public :: mesh_create, mesh_create_incore, mesh_repartition, &
          mesh_visualize, edge_migration, node_migration
integer, parameter :: mesh_dim=2, nodes_=3, edges_=3, neigh_=3
type element
     private
     integer :: id, nodeix(nodes_), edgeix(edges_), neighix(neigh_)
end type element
type mesh
     private
     type(node), dimension(:), pointer :: nodes
     type(edge), dimension(:), pointer :: edges
     type(element), dimension(:), pointer :: elements
     type(r_indx), dimension(:), pointer :: boundary_elements
end type mesh
contains
     subroutine mesh_create_incore(this, mesh_file)
     type (mesh), intent(inout) :: this
     character(len=*), intent(in) :: mesh_file
          ! details omitted...
     end subroutine mesh_create_incore
end module mesh_module
```

Figure 4. Skeleton view of mesh_module components.

tains statement, such as mesh_create_incore(), belong to the module. This means that routine interfaces, that perform type matching on arguments for correctness, are created automatically. (This is similar to function prototypes in other languages.)

**Distributed structure organization**

When the PAMR mesh data structure is constructed it is actually distributed across the processors of the parallel machine. This construction process consists of loading the mesh data, either from a single processor for parallel distribution (in_core) or from individual processors in parallel (out_of_core). A single mesh_build routine is responsible for constructing the mesh based on the data provided. Fortran 90 routine overloading and optional arguments allow multiple interfaces to the mesh_build routine, supporting code reuse. This is helpful because the same code that builds a distributed PAMR mesh data structure from the initial description can be applied to rebuilding the data structure after refinement and mesh migration. The mesh_build routine, and its interface, is hidden from public use. Heap sorting techniques are also applied in building the hierarchical structure so that reconstruction of a distributed mesh after refinement and mesh migration can be performed from scratch, but efficiently.

The main requirement imposed on the distributed structure is that every element knows its neighbors. Local neighbors are easy to find on the current processor from the PAMR structure. Remote neighbors are known from the boundary_elements section of the mesh data structure, in combination with a neighbor indexing scheme. When an element must act on its neighbors the neighbor index structure will either have a reference to a complete description of the local neighbor element or a reference to a processor_id/global_id pairing. This pairing can be used to fetch any data required regarding the remote element neighbor. (Note that partition boundary data, such as a boundary face in three-dimensions, is replicated on processor bound-

aries.) One of the benefits of this scheme is that any processor can refer to a specific part of the data structure to access its complete list of non-local elements.

Figure 4 showed the major components of the mesh data structure, in two-dimensions. While Fortran 90 fully supports linked list structures using pointers, a common organization for PAMR codes, our system uses pointers to dynamically allocated arrays instead. There are a number of reasons why this organization is used. By using heap sorting methods during data structure construction, the array references for mesh components can be constructed very quickly. Pointers consume memory, and the memory references can become "unorganized", leading to poor cache utilization. While a pointer-based organization can be useful, we have ensured that our mesh reconstruction methods are fast enough so that the additional complexity of a pointer-based scheme can be avoided.

**Interfacing among data structure components**

The system is designed to make interfacing among components very easy. Usually, the only argument required to a PAMR public system call is the mesh itself, as indicated in Figure 3. There are other interfaces that exist however, such as the internal interfaces of Fortran 90 objects with MPI and the ParMeTiS parallel partitioner [8] which written in the C programming language.

Since Fortran 90 is backward compatible with Fortran 77 it is possible to link to MPI for interlanguage communication, assuming that the interface declarations have been defined in the mpi.h header file properly. While certain array constructs have been useful, such as array syntax and subsections, MPI does not support Fortran 90 directly so array subsections cannot be (safely) used as parameters to the library routines. Our system uses the ParMeTiS graph partitioner to repartition the mesh for load balancing. In order to communicate with ParMeTiS our system internally converts the distributed mesh into a distributed graph. A single routine interface to C is created that passes the graph description from Fortran 90 by reference. Once the partitioning is complete, this same interface returns from C an array that describes the new partitioning to Fortran 90. This is then used in the parallel mesh migration stage to balance mesh components among the processors.

```
subroutine mesh_repartition(this)
type (mesh), intent(inout) :: this
   ! statements omitted...
   call PARMETIS(mesh_adj, mesh_repart, nelem, nproc, iam) ! C call
   call mesh_build(this, new_mesh_repart=mesh_repart)
end subroutine mesh_repartition
```

Figure 5. Fortran 90/C interface to mesh repartitioner and mesh migration routines

### 3.2 Parallel Mesh Migration and Load Balancing

Once the mesh is refined load imbalance is introduced, due to the creation of new elements in regions with high error estimates. As a result, the elements must be repartitioned and migrated to the proper processors to establish a balanced load. The ParMeTiS graph partitioner is used to compute the new partitioning. Elements are weighted based on the refinement level, the dual-graph of the mesh is created, and the ParMeTiS partitioner computes a new partitioning based on the weighted graph. The weighted graph attempts to find a partitioning that minimizes the movement of elements and the number of components on partition boundaries (to minimize communication). Our system, in order to minimize communication even further, actually gives ParMeTiS a mesh that only indicates refinement, where the new elements have not yet been created. Once this (coarse) mesh is migrated then the actual refinement is performed after element migration.

**Interfacing among C and Fortran 90 for mesh migration**

ParMeTiS only returns information on the mapping of elements to (new) processors, it does not actually migrate elements across a parallel system. Our parallel mesh migration scheme reuses the efficient mesh_build() routine to construct the new mesh from the ParMeTiS repartitioning. During this mesh_build process the element information is migrated according to this partitioning.

As seen in Figure 5, information required by the ParMeTiS partitioner is provided by calling a Fortran 90 routine that converts the mesh adjacency structure into ParMeTiS format (hidden). When this call returns from C, the private mesh_build() routine constructs the new distributed mesh from the old mesh and the new repartitioning by performing mesh migration. Fortran 90 allows optional arguments to be selected by keyword. This allows the mesh_build routine to serve multiple purposes since a keyword can be checked to determine if migration should be performed as part of the mesh construction process:

```
subroutine mesh_build(this, mesh_file, new_mesh_repart, in_core)
integer, dimension(:), intent(in), optional :: new_mesh_repart
logical, intent(in), optional :: in_core
! statements omitted...
   if (present(new_mesh_repart)) then
      ! perform mesh migration...
   end if
   ! (re)construct the mesh independent of input format...
end subroutine mesh_build
```

This is another way in which the new features of Fortran 90 add robustness to the code design. The way in which the new mesh data is presented, either from a file format or from a repartitioning, does not matter. Once the data in organized in our private internal format the mesh can be reconstructed by code reuse.

**Mesh migration communication algorithm**

The mesh is migrated in stages, based on the component type, for safety. In two-dimensions, mesh edges, nodes, and node coordinates are transported to new processors (if necessary) in that order. Since the mesh is reconstructed by the mesh_build() routine, information regarding the boundaries and component ownership does not need to be included in the migration stage.

The parallel communication algorithm for migration of mesh data is straightforward. Processors first organize data that will remain local. Then, data that must be migrated is sent continually to processors that expect the data. While the sending is performed, processors probe for incoming messages, that are expected, and receive them immediately upon arrival (the probe is non-blocking). Probing has the added benefit that a processor can allocate storage for an incoming message before the message is actually received. When this process is completed, processors check to see if there are any remaining receives, processes them if necessary, and the migration completes. At this point, the mesh is reconstructed with the new data.

The ParMeTiS library determines where elements must be migrated, using a multi-level diffusion algorithm. In Figure 6 we see a mesh with a random distribution of elements and the repartitioning after mesh migration using ParMeTiS.

### 3.3 Performance

The performance of the communication intensive parts of the system, such as mesh refinement, mesh migration, mesh loading, and mesh construction are of interest. The quality of the partitioning produced by

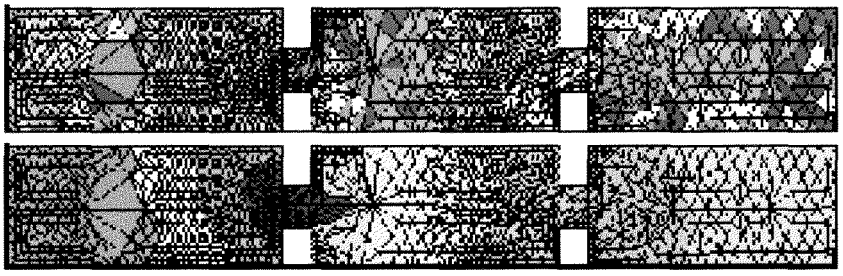

Figure 6. Illustration of ParMeTiS repartitioning on Cray T3E using 8 processors.

the ParMeTiS mesh partitioner, as well as its performance are also important. These features will be characterized and included in the final version of this abstract.

The element quality due to successive adaptive refinement could degrade rapidly to make the resulting mesh practically useless for many numerically applications. We therefore have incorporated a technique that improves the adaptive refinement process. Figure 7 shows a test case illustrating how narrow "green-refined" elements have been replaced by elements with better aspect ratios. (The improved AMR code will be applied to the waveguide filter and other examples shown here in our final paper).

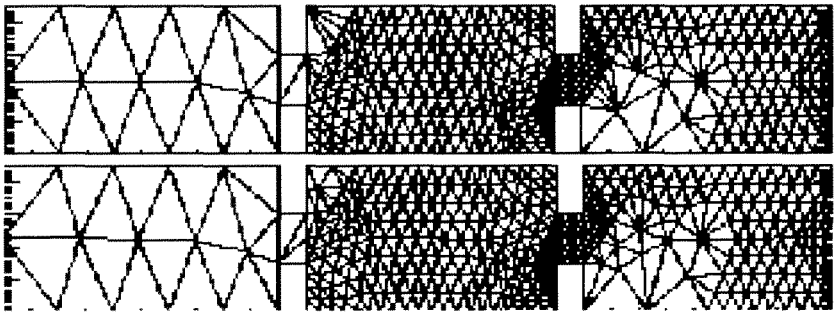

Figure 7. Illustration of mesh quality control during repeated adaptive refinement.

## 4. Applications

We now present some results from applications of our parallel AMR tool to a few test problems on triangular and tetrahedral meshes. Figure 6 shows a parallel partitioning and migration of a triangular finite-element mesh in a waveguide filter domain. The input mesh is read in from a disk file, and initially distributed in a random fashion on eight Cray T3E processors. The mesh is concurrently partitioned using the ParMeTiS routine. The parallel mesh migration module is then used to move subpartitions to their destination processors.

Our AMR module is tested in a finite-element simulation of electromagnetic wave scattering in the above waveguide filter [5]. The problem is to solve Maxwell's equation for the electromagnetic (EM) fields in the filter domain. A local-error estimate procedure based on the Element Residue Method (ERM) [1] is used in combination with the AMR technique to adaptively construct an optimal mesh for the problem solution. Figure 8 shows a few snapshots of the mesh in the AMR solution process. The color and density distribution of mesh elements in the figure reflect the (estimated) error distribution in the computed fields. Another application of the AMR module is to an EM simulation in a quantum well infrared photodetector (QWIP), as shown in Figure 9.

Figure 10 shows a test of our AMR module on a tetrahedral mesh. The initial tetrahedral mesh

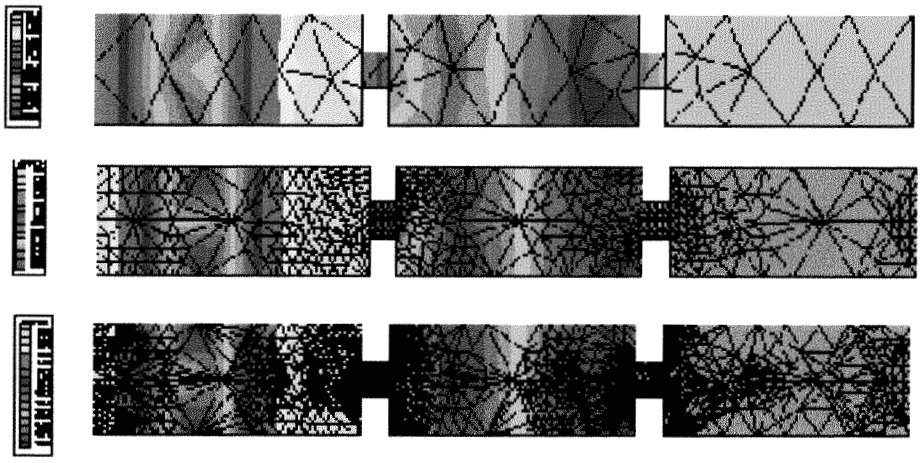

Figure 8. Adaptive finite-element solution in a waveguide filter. Adaptive refinement is guided by a local-error estimate procedure based on local residuals.

was generated in a U-shaped domain with 120 elements. Mesh elements in two spherical subregions, indicated by the circles in the top initial mesh, are chosen for adaptive refinement. The radius of the refining spheres is reduced by 20% after each adaptive refinement. The color image at the bottom of Figure 10 is the resulting mesh after three successive adaptive refinements, which has about 2500 elements.

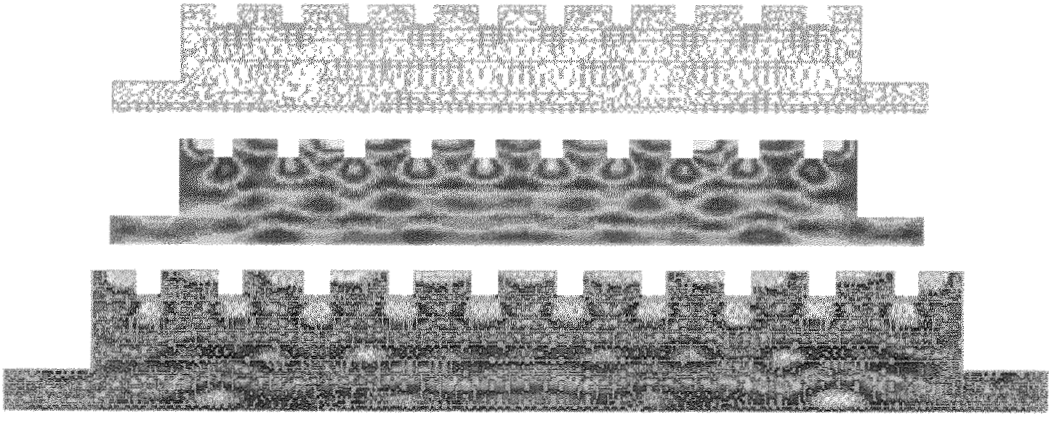

Figure 9. Adaptive finite-element simulation in a quantum well infrared photodetector for long-wavelength infrared radiation. The adaptively refined mesh, computed magnetic field relative to an incident plane wave, and the wave field on the mesh are shown respectively.

## 5. Conclusion

We have presented a complete framework for performing parallel adaptive mesh refinement in unstructured applications on multiprocessor computers. A robust parallel AMR scheme and its implementation with mesh quality control, as well as a load-balancing strategy in parallel AMR, are discussed. Our implementation of the parallel AMR software package in Fortran 90 and MPI, including the data structure and interfaces between different modules, are also discussed. A few application examples using our developed AMR modules are demonstrated. Parallel performance on several multiprocessor systems will be given in our

final paper.

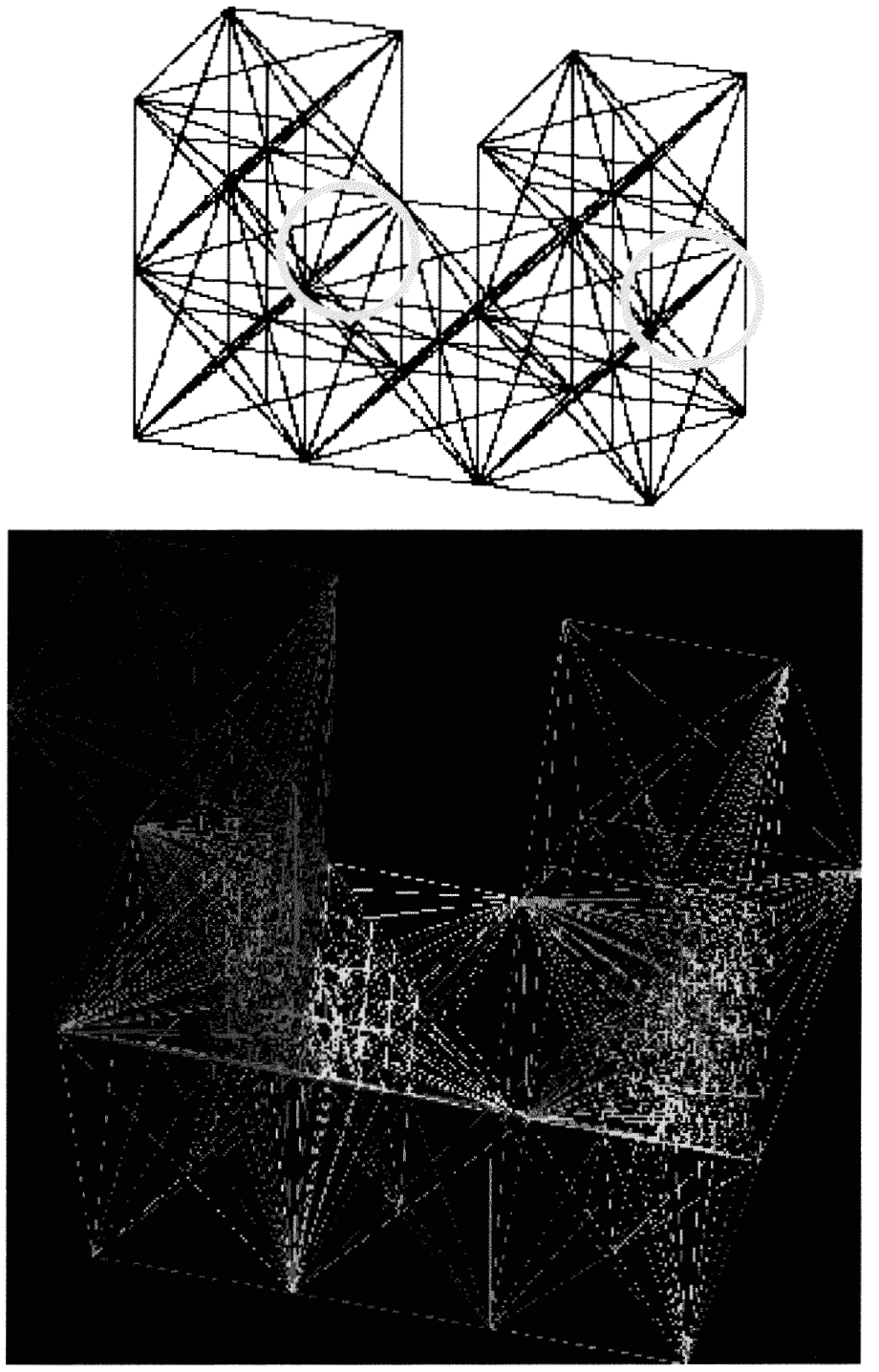

Figure 10. Adaptive refinement on a three-dimensional tetrahedral mesh. The initial mesh (top) has 128 elements, and two subregions are chosen arbitrarily to be refined. The mesh after adaptive refinements (bottom) has about 2,500 elements.

## 6. References


[1] M. Ainsworth, J. T. Oden. A Procedure for a Posteriori error Estimation for *h-p* Finite Element Methods." Computer Methods in Applied Mechanics and Engineering, 101 (1972) 73-96.

[2] R. Biswas, L. Oliker, and A. Sohn. "Global Load-Balancing with Parallel Mesh Adaption on Distributed-Memory Systems." Proceedings of Supercomputing '96, Pittsburgh, PA, Nov. 1996.

[3] E. Boender. "Reliable Delaunay-Based Mesh Generation and Mesh Improvement." Communications in Numerical Methods in Engineering, Vol. 10, 773-783 (1994).

[4] Graham F. Carey, "Computational Grid Generation, Adaptation, and Solution Strategies". Series in Computational and PHysical Processes in Mechanics and Thermal Science. Taylor & Francis, 1997.

[5] T. Cwik, J. Z. Lou, and D. S. Katz, "Scalable Finite Element Analysis of Electromagnetic Scattering and Radiation." to appear in Advances in Engineering Software, V. 29 (2), March, 1998

[6] V. Decyk, C. Norton, and B. Szymanski. Expressing Object-Oriented Concepts in Fortran 90. ACM Fortran Forum, vol. 16, num. 1, April 1997.

[7] L. Freitag, M. Jones, and P. Plassmann. "An Efficient Parallel Algorithm for Mesh Smoothing." Tech. Report, Argonne National Laboratory.

[8] G. Karypis, K. Schloegel, and V. Kumar. "ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 1.0". Tech. Rep., Dept. of Computer Science, U. Minnesota, 1997.

[9] C. Norton, V. Decyk, and B. Szymanski. High Performance Object-Oriented Scientific Programming in Fortran 90. Proc. Eighth SIAM Conf. on Parallel Processing for Sci. Comp., Mar. 1997 (CDROM).

[10] M. Shephard, J. Flaherty, C. Bottasso, H. de Cougny, C. Ozturan, and M. Simone. Parallel automatic adaptive analysis. Parallel Computing 23 (1997) pg. 1327-1347.